

Analysis of Equality Relationships for Imperative Programs

Pavel Emelianov*
Institute of Informatics Systems,
6 avenue Lavrentiev
630090 Novosibirsk, Russia
emelianov@iis.nsk.su

February 1, 2008

Abstract

In this article, we discuss a flow-sensitive analysis of equality relationships for imperative programs. We describe its semantic domains, general purpose operations over abstract computational states (term evaluation and identification, semantic completion, widening operator, etc.) and semantic transformers corresponding to program constructs. We summarize our experiences from the last few years concerning this analysis and give attention to applications of analysis of automatically generated code. Among other illustrating examples, we consider a program for which the analysis diverges without a widening operator and results of analyzing residual programs produced by some automatic partial evaluator. An example of analysis of a program generated by this evaluator is given.

Keywords: abstract interpretation, value numbering, equality relationships for program terms, formal grammars, semantic transformers, widening operator, automatically generated programs.

Introduction

Semantic analysis is a powerful technique for building effective and reliable programming systems. In [17, 18, 16] we presented a new kind of a semantic flow-sensitive analysis designed in the framework of *abstract interpretation* [9, 10, 11]. This analysis which determines an approximation of sets of invariant term equalities $t_1 = t_2$ was called *the analysis of equality relationships for program terms* (hereinafter referred to as **ERA**).

Most traditional static analyses of imperative programs are interested in finding the (in)equalities of a specific kind (so-called *value analyses*; only they are discussed here)

*This work was partly done when the author was in *Laboratoire d'informatique, Ecole polytechnique* (Palaiseau, France) and *Ecole normale supérieure d'ingénieur* (Bourges, France).

describing regular approximations (i.e. they have simple mathematical descriptions and machine representations) of sets of values: convex polyhedrons/octahedrons/octagons [14, 27, 36, 8], affine [32, 25] and congruent [24, 34] hyper-planes, their non-relational counterparts [9, 43, 23, 35] as well, etc. They are carefully designed to be reasonable (i.e. they express non-trivial semantic properties) and effectively computed (i.e. there are polynomial algorithms to handle them¹) but “regular” nature does not allow them to treat well programs with irregular control/data-flows. Hence of special interest is investigations of approximations based on sets of terms which can have potentially arbitrary nature, i.e. they could be powerful (due to their irregularity) but effectively computed. One well known example is the set-based analysis [29, 12].

In our case, terms represent all expressions computed in programs. This enables the analysis to take into account different aspects of program behavior in a unified way. A such unified treatment of all semantic information allows the analysis to improve its accuracy. This does not mean that **ERA** is a generalization of all other value analyses (except the *constant propagation* one), because they use different approaches (semantic domains and transformers) to extract effectively and precisely the limited classes of semantic properties. In general, the results of the analyses are not comparable.

ERA provides interesting possibilities for gathering and propagating different invariant information about programs in a unified way. This information can be used both for verification and optimization purposes. The second is especially interesting for automatically generated programs: residual, i.e. obtained in the process of the partial evaluation, and synthesized from high-level specifications. Due to nature of automatic generation processes, such programs have specific control flows (for example, hierarchy of nested conditional statements with specific conditions; in the case of residual programs this hierarchy is more deep as “degree” of the partial evaluation increases) that can be successfully optimized on base of gathered invariant information.

Besides the peculiarity of **ERA** mentioned above, let us discuss some common properties of the semantic analyses. Such taxonomic properties of the analysis algorithms as the attribute (in)dependence, context (in)sensitivity, flow (in)sensitivity, scalability and some other properties are well known. However, it is the author’s opinion that a notion of “*interpretability of a semantic analysis*” has not been considered adequately yet. Here the interpretability of analysis means how extensively the properties of primitive operations of the language (arithmetical, logical, etc.) and type information are allowed for analyzing and can be handled when the analysis works.

One extreme point of view on the interpretability is an approach accepted in the “pure” program scheme theory where no interpretations of functional symbols or type information are allowed². Unfortunately, the results obtained under this approach are not reasonably strong. Nevertheless, it must be underscored that **ERA** dates back to V. Sabelfeld’s works in the program scheme theory [39, 40]. Another extreme leads to the complete description of the program behavior that is also not workable. Obviously, it is closely allied to its flow sensitivity (ignoring some part of semantic information

¹An example of this sort is an approximation of value sets by conic shapes that has only one “computational disadvantage”: semantic transformers involve algorithms known to be \mathcal{NP} -hard. Proposed in [4] many years ago it did not gain ground.

²More precisely, there exist some works in the program scheme theory where some semantic interpretations of functional symbols (like commutativity of superposition $f \circ g \equiv g \circ f$, etc.) are considered.

does not allow us to treat precisely some control flow constructs of analyzed programs) and its scalability (attempts to take into account large quantity of semantic properties, for example, using some theorem prover which is invoked while a semantic analyzer works and deduces new properties, can lead to combinatorial explosion for abstract computational states).

It is possible that the interpretability has not been highlighted enough, because most of analysis algorithms take into account the limited classes of primitive operations and type information and they cannot be enriched in some natural way. For example, an interval analysis is not able to incorporate congruence properties in some natural way, etc³. Essentially another case is **ERA** where we have a choice to handle expressiveness of the analysis. We intend to illustrate the notion of “interpretability of analysis”, its importance and usefulness on this example of analysis.

Among the analyses closely related to ours we would like to point out the following. A semantic analysis for detection of equalities between program variables (and simple relationships among them) was described in [2]. It makes a list of sets of variables discovered to be equal by using the Hopcroft’s partitioning algorithm for finite-state automata. This algorithm being quite efficient is not however precise enough. Further value-numbering techniques were developed in [38, 21, 26]. These algorithms demonstrate that adequacy of value numbering is resource-consuming. For example, in the last case time complexity of the algorithm is $O(k^3 j N)$ where k is a number of program variables, j is a number of join-points, and N is program size.

Another important example is the set-based analysis [29] mentioned above. Here approximating sets of terms are found with resolving some system of set-theoretical equations. Formal grammars were used for an analysis of recursive data structures of functional languages (see, for example, [31]). Formal languages were applied to coding of memory access paths in [15, 42] and values of program variables in the set-based analysis. [12] established common foundations connecting and generalizing different approaches using formal languages to represent semantic properties. Of course, we should mention techniques from the automatic proof theory and the term rewriting theory which can be widely applied both at the analysis stage to improve its accuracy and the post-processing stage to present its results to the user.

This article is organized as follows: In **Sections 1.1** and **1.2** we describe the semantic properties, concrete and abstract, respectively, which are considered in **ERA**. In **Section 1.3** we discuss some basic operations over the semantic properties used to define the semantic transformers which are next presented in **Section 2**. In **Section 3** we consider a widening operator and the complexity of **ERA** is discussed. Finally, **Section 4** describes processing of **ERA** invariants and presents some results of our experiments with **ERA**. In **Appendix** an example of analysis of some residual program is considered.

³Of course, it is possible to use sophisticated approaches for combinations of analyses but it introduces complicated problems under implementations and it is not an enrichment of original ones.

1 Properties of interest

1.1 Concrete properties

A usual choice for the description of the operational semantics is a specification of some transition relationship on the pairs $\langle \text{control point}, \text{state of program memory} \rangle$ (see, for example, [30, 22]) where the states of program memory are described by mapping the cells of memory into a universe of values. Here variables (groups of cells) and their values (constants) are in asymmetric roles. Another example of “asymmetry”: manipulations over the structured objects of programs (arrays, records etc.) are not so transparent as over the primary ones. To describe the operational semantics for **ERA**, we used another approach. All objects of a program are considered to be “identical” in the following meaning.

Let \mathcal{CV} be a set of 0-ary symbols representing variables and constants. The last ones may be of the following kinds: scalars, compositions over scalars (i.e., constant arrays, records, etc.), names of record fields, and *indefiniteness*. Let \mathcal{FP} be a set of n-ary (*functional*) symbols which represent primitive operations of programming languages: arithmetic, logic, type casting, and all the kinds of memory addressing, as well. Let \mathcal{TS} be a set of well-formed terms over \mathcal{CV} and \mathcal{FP} , hereinafter referred to as *program terms*. They represent expressions computed during execution of a program. So, as a state of program memory we take a reflexive, symmetrical, and transitive relationship (i.e., *equivalence relationship*) over \mathcal{TS} . The relationship defines some set of term equalities which we use to describe the operational semantics and call a *computation state*.

Suppose that the following code

```
var x,i,j : integer; a : array [1..3] of integer={1,2,3};
...
i := 3;
j := i-1;
if odd(x) then
  i := i mod j;
  j := 1;
else
  j := a[i];
  a[i] := a[1];
  a[1] := j
end
...
```

Example 1

is executed at least twice for the different parities of the variable x. In **Table 1** static semantics for five control points is given. We present a minimum⁴ subset of term equalities concerning dynamic behavior of the piece. We shall use the property π (see **Table 1**) to illustrate our further reasoning.

⁴A set of equalities can be completed with any number of consistent equalities.

	THEN-branch	ELSE-branch
ENTRY	$\left\{ \left\{ \begin{array}{l} a[1]=1, a[2]=j=2, a[3]=i=3, \\ a=\boxed{1 \ 2 \ 3}, \text{ODD}(x)=\text{TRUE} \end{array} \right\} \right\}$	$\left\{ \left\{ \begin{array}{l} a[1]=1, a[2]=j=2, a[3]=i=3, \\ a=\boxed{1 \ 2 \ 3}, \text{ODD}(x)=\text{FALSE} \end{array} \right\} \right\}$
EXIT	$\left\{ \left\{ \begin{array}{l} a[1]=i=j=1, a[2]=2, a[3]=3, \\ a=\boxed{1 \ 2 \ 3}, \text{ODD}(x)=\text{TRUE} \end{array} \right\} \right\}$	$\left\{ \left\{ \begin{array}{l} a[1]=i=j=3, a[2]=2, a[3]=1, \\ a=\boxed{3 \ 2 \ 1}, \text{ODD}(x)=\text{FALSE} \end{array} \right\} \right\}$
EXIT OF IF-STATEMENT		
π	$= \left\{ \left\{ \begin{array}{l} a[1]=i=j=1, a[2]=2, a[3]=3, \\ a=\boxed{1 \ 2 \ 3}, \text{ODD}(x)=\text{TRUE} \end{array} \right\}, \left\{ \begin{array}{l} a[1]=i=j=3, a[2]=2, a[3]=1, \\ a=\boxed{3 \ 2 \ 1}, \text{ODD}(x)=\text{FALSE} \end{array} \right\} \right\}$	

Table 1: Description of collecting semantics for **Example 1** (here the constant $\boxed{c_1 \ c_2 \ c_3}$ represents constant arrays).

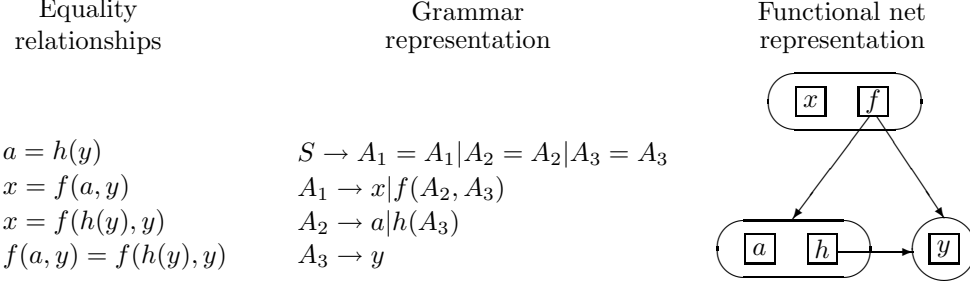


Figure 1: Semantic properties and their representations. The set of equalities does not contain trivial equalities like $x = x$ and equalities given by the symmetry of the equality relationship.

Formally it is described as follows. Let \mathcal{EQS} be a set of all equalities of the terms from \mathcal{TRS} , i.e., $\mathcal{EQS} = \{t_1 = t_2 \mid t_1, t_2 \in \mathcal{TRS}\}$. A set $S \in \wp(\mathcal{EQS})$ is a computation state interpreted in the following way. For each equality $t_i = t_j \in S$ values of the expressions represented by t_i and t_j must be equal at this point for some execution trace. We take the set $\wp(\wp(\mathcal{EQS}))$ as a set for a concrete semantic domain describing *the collecting semantics* of **ERA**. So, an element of the concrete semantic domain is a set of computation states. For a particular point in a particular program it is a set of computation states each of them corresponds to some execution trace in the program that reaches this point.

Properties considered in **ERA** are presented by means of context-free grammars $\mathbf{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, S)$ where \mathcal{N} is a finite set of nonterminals denoted by capital letters, $S \in \mathcal{N}$ is the initial symbol of the grammar \mathbf{G} , $\mathcal{T} = \mathcal{CV} \cup \mathcal{FP} \cup \{ (,), =, \}$ is a finite set of terminal symbols, and \mathcal{P} is a finite set of grammar rules. We do not give their precise description because we shall use quite simple machinery from formal languages theory that is not an object of considerations itself and serves for demonstrations. We could use functional nets language as well but it is rather machine-oriented and is not widely used. We expect that all these descriptive ways become apparent from the examples on **Figure 1** and further ones.

In that way, a state of computation is represented by a language $L(\mathbf{G})$ generated by some grammar \mathbf{G} of the described form. If for $A \in \mathcal{N}$: $A \xrightarrow{\pm}_Q t_1 \wedge A \xrightarrow{\pm}_Q t_2$, i.e. $t_1 = t_2 \in L(\mathbf{G})$, we say that the nonterminal A and the language $L(\mathbf{G})$ know the terms t_1, t_2 . Obviously, such a grammar representation has some superfluous “syntactic sugaring”. We can use $S \rightarrow A$ rules only and say about A -nonterminals as classes of equal values.

Evidently we may suppose that the set of rules \mathcal{P} does not contain rules having identical right parts. It is convenient to consider the grammars which do not contain useless and redundant nonterminals and rules. A rule is useless if it produces only one term (the language knows only a trivial equality like $t = t$) and this term is not an argument of other terms. A nonterminal is useless if it does not participate in derivations

of sentential forms [1]. Such grammars can arise as result of operations on grammars. If nonterminals and rules are useless or redundant it is possible to remove them (see **Lemma 2**). This operation called **state reduction** consist in detecting and removing a set of useless/redundant nonterminals/rules. They are revealed with the well-known incremental markup algorithms (see, for example, [1]).

For illustrating purposes we shall use special functional nets for these grammars. Here nonterminals are represented by ovals containing 0-ary and functional symbols from right parts of rules. Arcs from functional symbols to ovals represent argument dependencies ordered from left to right (see **Figure 1**).

1.2 Abstract properties

It is an interesting peculiarity of **ERA** that abstract (i.e. approximate) properties have the same nature as the computation states of the operational semantics. Formally this approximation is defined as follows.

1.2.1 Functions of abstraction and concretization

Given a concrete property $\pi \in \wp(\wp(\mathcal{E}\mathcal{Q}\mathcal{S}))$ and an abstract property $\tilde{\pi} \in \wp(\mathcal{E}\mathcal{Q}\mathcal{S})$, the abstraction function $\alpha : \wp(\wp(\mathcal{E}\mathcal{Q}\mathcal{S})) \rightarrow \wp(\mathcal{E}\mathcal{Q}\mathcal{S})$ and the concretization one $\gamma : \wp(\mathcal{E}\mathcal{Q}\mathcal{S}) \rightarrow \wp(\wp(\mathcal{E}\mathcal{Q}\mathcal{S}))$ are defined in the following way

$$\alpha(\pi) = \begin{cases} \mathcal{E}\mathcal{Q}\mathcal{S}, & \text{if } \pi = \emptyset, \\ \bigcap_{S \in \pi} S & \text{otherwise} \end{cases} \quad \text{and} \quad \gamma(\tilde{\pi}) = \bigcup \{ \pi \mid \alpha(\pi) \supseteq \tilde{\pi} \}$$

where \cup is the set-theoretical union on $\wp(\wp(\mathcal{E}\mathcal{Q}\mathcal{S}))$ and \supseteq and \cap are the set-theoretical inclusion and intersection of the languages (i.e. on $\wp(\mathcal{E}\mathcal{Q}\mathcal{S})$) respectively. We take the empty language as infimum \perp' (there are no computed expressions) of the semi-lattice of abstract semantic properties. The supremum \top' (an inaccessible computation state) is the language containing all possible equalities of program terms. Also \subseteq is the set-theoretical inclusion on $\wp(\wp(\mathcal{E}\mathcal{Q}\mathcal{S}))$

Lemma 1 *The abstraction function α is monotonic.*

Proof The function α is monotonic iff $\forall \pi_1, \pi_2 \in \wp(\wp(\mathcal{E}\mathcal{Q}\mathcal{S})) : \pi_1 \subseteq \pi_2 \Rightarrow \alpha(\pi_1) \supseteq \alpha(\pi_2)$. Because $\pi_1 \subseteq \pi_2$, then $\pi_2 = \pi_1 \cup (\pi_2 \setminus \pi_1)$. So, we have $\alpha(\pi_1) \supseteq \alpha(\pi_1 \cup (\pi_2 \setminus \pi_1)) = \alpha(\pi_1) \cap \alpha(\pi_2 \setminus \pi_1) = \alpha(\pi_1) \cap (\bigcap_{S \in \pi_2 \setminus \pi_1} S)$. Q.E.D.
 $\gamma(\tilde{\pi})$ is the most imprecise element of $\wp(\wp(\mathcal{E}\mathcal{Q}\mathcal{S}))$ that can be soundly approximated by $\tilde{\pi} \in \wp(\mathcal{E}\mathcal{Q}\mathcal{S})$. For the example of **Table 1**, the best approximation of the concrete property π is

$$\alpha(\pi) = \left\{ a[1] = i = j, a[2] = 2 \right\}. \quad (*)$$

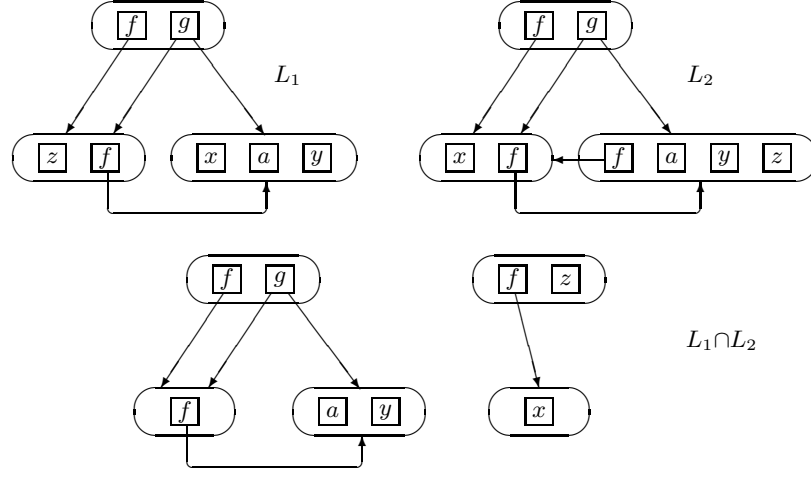


Figure 2: Intersection of computation states.

1.2.2 Intersection of ERA-languages

Finding of the intersection of context-free languages is an undecidable problem in the general case. In our case, for the languages of term equalities, an algorithm exists (see an example at **Fig. 2**). It is similar to constructing a Cartesian product of automata.

Algorithm. Intersection of two languages of term equalities.

Input: grammars $\mathbf{G}_1 = (\mathcal{N}_1, \mathcal{T}, \mathcal{P}_1, S_1)$ and $\mathbf{G}_2 = (\mathcal{N}_2, \mathcal{T}, \mathcal{P}_2, S_2)$.

Output: grammar $\mathbf{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, S)$ such that $L(\mathbf{G}) = L(\mathbf{G}_1) \cap L(\mathbf{G}_2)$.

Description:

1. Let $\mathcal{N} = \{\langle N_1, N_2 \rangle \mid N_1 \in \mathcal{N}_1 \text{ \& } N_2 \in \mathcal{N}_2\} = \mathcal{N}_1 \times \mathcal{N}_2$.
2. The set of rules \mathcal{P} is defined as follows:
 - The rule $\langle N_1, N_2 \rangle \rightarrow t$ is introduced if and only if $t \in \mathcal{CV}$ \& $N_1 \rightarrow t \in \mathcal{P}_1$ \& $N_2 \rightarrow t \in \mathcal{P}_2$.
 - The rule $\langle N_1, N_2 \rangle \rightarrow t(\langle N_1^1, N_2^1 \rangle, \dots, \langle N_1^k, N_2^k \rangle)$ is introduced if and only if $t \in \mathcal{FP}$ \& $N_1 \rightarrow t(N_1^1, \dots, N_1^k) \in \mathcal{P}_1$ \& $N_2 \rightarrow t(N_2^1, \dots, N_2^k) \in \mathcal{P}_2$.
3. Add rules $S \rightarrow \langle N_1, N_2 \rangle = \langle N_1, N_2 \rangle$ for the initial nonterminal S of \mathbf{G} and for all $N_1 \in \mathcal{N}_1 \setminus \{S_1\}, N_2 \in \mathcal{N}_2 \setminus \{S_2\}$.
4. Apply *state reduction*.

The described algorithm of intersection is very “naïve” and impractical. To improve it we should choose a more efficient strategy for generating functional symbols. To do this we first do a topological sorting of the functional symbols appearing in the right

parts of rules; intersect the 0-ary symbol sets; and then generate the next functional symbol in conformity with its topological order if and only if all arguments of this symbol already exist in the new grammar. For practical cases this intersection can be done in an linear average time with respect to grammar size and linear space. It demands quadratic time in the worst case.

1.3 Operations over semantic properties

Now we shall discuss some basic operations over semantic properties $\{\cdot\} : \wp(\mathcal{EQS}) \rightarrow \wp(\mathcal{EQS})$ used to define the semantic transformers of **ERA**. For all $\{\cdot\}$ the notation $L\{\cdot\}\{\cdot\}$ means $(L\{\cdot\})\{\cdot\}$.

1.3.1 Removing terms

Operations over abstract computation states use certain common transformation of the sets of term equalities which consists in **removing** some subset L' . The following statement holds.

Lemma 2 *Removing any subset of term equalities preserves correctness of an approximation.*

Proof It easy to see that

$$\begin{aligned} \cup\{ \pi \mid \alpha(\pi) \supseteq \tilde{\pi} \} = \gamma(\tilde{\pi}) \quad \subseteq \quad \gamma(\tilde{\pi} \setminus L') = \cup\{ \pi \mid \alpha(\pi) \supseteq (\tilde{\pi} \setminus L) \} = \\ \cup\{ \pi \mid \bigcap_{L \in \pi} (L \cup L') \supseteq \tilde{\pi} \}, \end{aligned}$$

which states that removing term equalities makes the approximation more rough but it does preserve its correctness. Q.E.D.

For $(*)$, for example, $\gamma(\{a[1] = i = j, a[2] = 2\}) \subseteq \gamma(\{a[1] = i\})$.

We shall write $L\{\downarrow t\}$ and $L\{\downarrow T\}$ for single term and term set removing followed by the state reduction operation defined above.

1.3.2 Term evaluation

We define the abstract semantics for an **evaluation of a term** in an abstract computation state. The result of the term evaluation is a state knowing the evaluated term.

Term evaluation $L\{\downarrow t\}$.

1. If $t = t \in L$, then $L\{\downarrow t\} = L$.
2. Otherwise, if $t \in \mathcal{CV}$, then add the new rules $S \rightarrow A = A$ and $A \rightarrow t$ to the grammar **G**, where A is a nonterminal which does not exist in **G**.

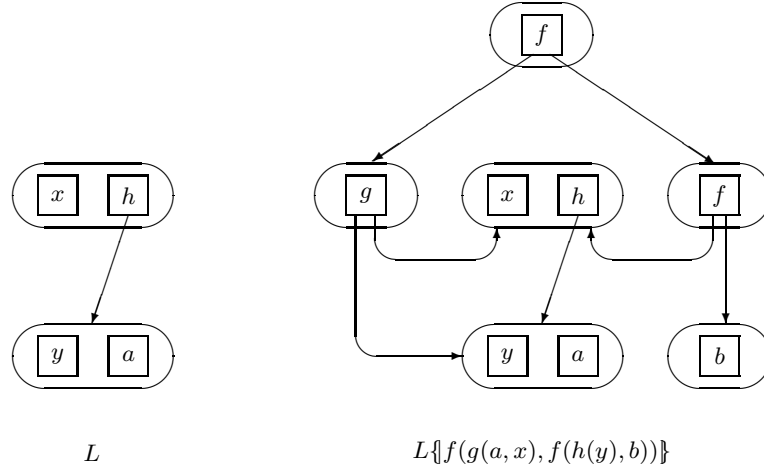


Figure 3: Term evaluation.

3. Otherwise, if $t = f(t_1, \dots, t_n)$ where $f \in \mathcal{FP}$ is a functional n -ary symbol and the sub-terms t_1, \dots, t_n have been calculated (i.e. there exist derivations $A_1 \xrightarrow{\pm}_{\mathcal{G}} t_1, \dots, A_n \xrightarrow{\pm}_{\mathcal{G}} t_n$), then add the new rules $S \rightarrow A = A$ and $A \rightarrow f(A_1, \dots, A_n)$ to the grammar \mathbf{G} , where A is a nonterminal which still does not exist in \mathbf{G} .

To improve an accuracy of analysis we can take into account, for example, the commutativity of primitive operations. If L knows $f(t_1, t_2)$ and f is commutative then $L\llbracket f(t_2, t_1) \rrbracket = L$.

1.3.3 Identification of terms

When the standard semantics defines that during program execution values of computed expressions are equal, we can incorporate this information in the computation state (but it can also be left out). **Identification of terms** transforms the state into a new one incorporating this information. For example, we know that a value of a term representing the conditional expression of an **IF**-statement coincides with 0-ary terms representing the constants **TRUE** or **FALSE** when, respectively, **THEN**-branch or **ELSE**-branch is being executed. So, identification of terms along with *semantic completion* considered below provides powerful facilities to take into account real control flow in programs.

Identification of terms $L\llbracket t_1 \equiv t_2 \rrbracket$.

1. If $t_1 = t_2 \in L$ then $L\llbracket t_1 \equiv t_2 \rrbracket = L$.
2. Let $A_1 \xrightarrow{\pm}_{\mathcal{G}} t_1$ and $A_2 \xrightarrow{\pm}_{\mathcal{G}} t_2$. We replace the nonterminal A_2 by the nonterminal A_1 in all rules of \mathcal{P} . If rules with an identical right side $B_1 \rightarrow w, \dots, B_k \rightarrow w$ have appeared, then a certain nonterminal from the left sides of the rules (for example B_1) must be taken and all nonterminals B_2, \dots, B_k in the grammar must be replaced by it.

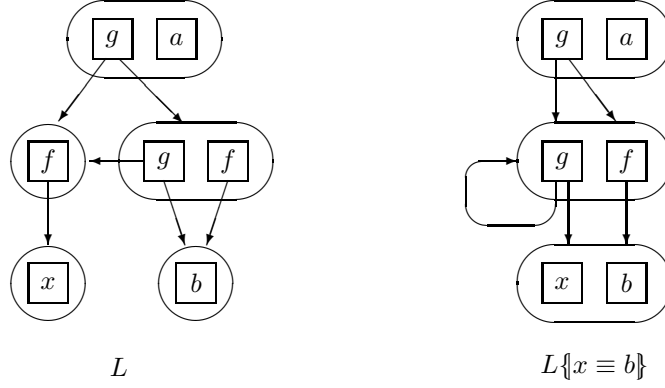


Figure 4: Identification of values of terms.

3. Repeat step 2 until stabilization. If after that we have a state L' containing inconsistent term equalities⁵ then the result is \top' else it is a reduction of L' .

An example of identification is given in **Figure 4**.

Lemma 3 *Identification of values of terms is a correct transformation and the resulting state is unique.*

Proof

Let $\pi = \{ L_i \mid L_i \in \wp(\mathcal{EQS}) \}$ be a concrete semantic property which holds before identification of terms t_1 and t_2 . If the values of t_1 and t_2 are equal in the concrete semantics $\forall L_i : t_1 = t_2 \in L_i$, then they are equal in the abstract semantics $t_1 = t_2 \in \alpha(\pi)$, too. If their values are not equal, then identification gives us an inconsistent computation state which obviously includes $L\{t_1 \equiv t_2\}$ for all t_1 and t_2 . So, this transformation is correct.

Identification is done in finite steps because the size of grammar decreases at each step. Uniqueness of the resulting state is explained by the following observation. If we have two pairs of terms which are candidates for identification, then identification of one of them does not close a possibility of it for another, because we remove a duplication of the functional symbols only. In fact, after identification of a pair of terms we obtain a new state, including the source one, and thus other existing identification possibilities remain. So, the order of “merging” of term pairs is not important for the resulting state. Q.E.D.

1.3.4 Semantic completion

We have not yet considered any interpretation of constants and functional symbols. We could continue developing **ERA** in the same way. As a result we shall obtain a

⁵There exists a wide spectrum of inconsistency conditions. The simplest of them is an equality of two different constants (see **Section 1.3.4** for the further discussion).

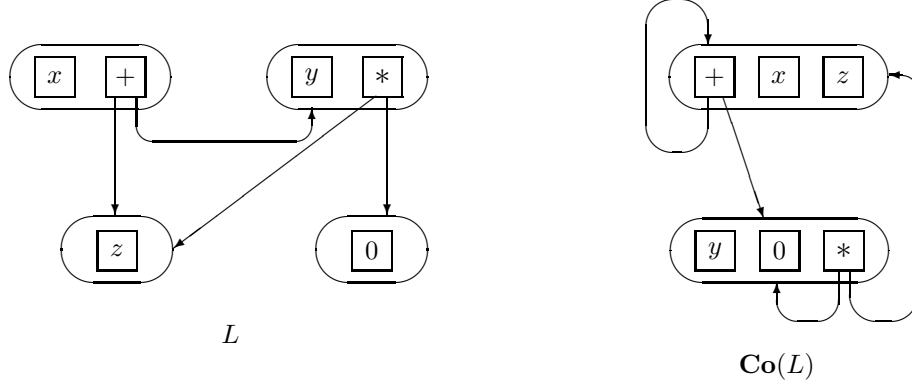


Figure 5: Semantic completion.

noninterpretational version of the analysis likewise analysis algorithms in the program scheme theory. However, it is natural to use semantics of primitive operations of the programming language of interest in order to achieve better accuracy.

ERA provides us with possibilities of taking into account properties of language constructs, and, what is especially important, we can easily handle complexity of these manipulations. In fact, inclusion of these properties corresponds to carrying out some finite part of completion of the computation states by consistent equalities. This manipulation is called **semantic completion** (about “conjunctive/disjunctive completion” see [11]).

As a basic version of **semantic completion Co** we take computations over constant and equal arguments. When we detect that some term t has some specific value v then $\mathbf{Co}(L) = L\{t \equiv v\}$. Also, it is possible to apply identification involving dependencies among result of an operation and its arguments: if $(t_1 \text{ AND } t_2) = \text{TRUE} \in L$ then $\mathbf{Co}(L) = L\{t_1 \equiv \text{TRUE}\}\{t_2 \equiv \text{TRUE}\}$ etc. This identification process is iterative because new possibilities for identification can appear at the next steps. It is conceivable that in doing so we shall detect inconsistency of a computation state. In this case result of semantic completion is \top' .

This version can be extended by some intelligent theorem prover inferring new reasonable equalities and checking inconsistency of computation states. Such combining of analysis with proofs offers powerful facilities to the analyzer (see [28, 13]) and, as it was mentioned in the previous works, this prover is reusable for consequent (semi-)automatic processing of results of the analysis. “Size” of used completion can be tuned by options of interpretability of the analyzer.

Some arithmetical errors (such as division by zero, out of type range etc.) will appear during the semantic completion. In this case the analyzer tells us about the error and sets the current computation state to \top' . Notice that for the languages where incomplete Boolean evaluation is admissible semantic completion over Boolean expressions should be carefully designed especially in presence of pointers.

An example of semantic completion is presented in **Figure 5**. Turning back to the identification example at **Figure 4**, we consider the following interpretation of constants

and functional symbols: g is the exclusive disjunction, f is the negation, a is the constant **TRUE** and b is the constant **FALSE**. It is easy to see that in this case application of semantic completion gives us $L\{x \equiv b\} = \top'$.

In our analyzer we implemented an interpretational version of **ERA** which uses semantic completion $\mathbf{Co}(L)$. Under this approach, the definitions of the basic transformations mentioned above are changed to the following:

$$\begin{aligned} L\{t\}' &= \mathbf{Co}(L\{t\}), \\ L\{t_1 \equiv t_2\}' &= \mathbf{Co}(L\{t_1 \equiv t_2\}). \end{aligned}$$

In short, we shall omit this “interpretability” prime.

2 Semantic transformers

In this section we describe semantic transformers $\llbracket \cdot \rrbracket : \wp(\mathcal{E}\mathcal{Q}\mathcal{S}) \rightarrow \wp(\mathcal{E}\mathcal{Q}\mathcal{S})$ corresponding to common statements existing in imperative programming languages. If for a statement **S** an input computation state is L then $L\llbracket S \rrbracket$ is its output computation state. For all $\llbracket \cdot \rrbracket$ the notation $L\llbracket \cdot \rrbracket \llbracket \cdot \rrbracket$ means $(L\llbracket \cdot \rrbracket)\llbracket \cdot \rrbracket$.

2.1 Assignment statement

Among all program terms considered in **ERA** we can pick out *access program terms* including array *elm*, record *fld*, and pointer *val* referencing and playing an important role in determination of effect of an assignment statement. As in [15, 42] our abstraction of program memory manipulations is storeless and based on notion of memory access paths represented by access program terms. For example, for an address expression **bar[i][j]^foo** the access term is $fld(val(elm(elm(bar, i), j)), foo)$.

We shall assume that no operations other than memory addressing (for example comparisons) are allowed for structured variables such as arrays and records. So, for the previous example neither a and $elm(a, i)$ (**a[i]**) nor $val(elm(elm(a, i), j))$ (**a[i][j]^**) can appear as arguments for operations other than *elm* and *fld* respectively. This limitation allows us to simplify definition of our assignment statement abstraction⁶.

To preserve safety of the analysis we have to take into account memory aliasing appearing in programs. Two access terms are alias if they address the same memory location. In the general case **ERA** is inadequate itself to handle precisely all kinds of aliasing and we should use other analyses. Next it is assumed that for each access term t_a we know a set $\mathcal{A}(t_a)$ of access terms covering a set of aliases for t_a (may-alias information about t_a). Let

$$\overline{\mathcal{A}(t_a, L(\mathbf{G}))} = (\mathcal{A}(t_a) \cap \mathcal{CV}) \cup \{f \in \mathcal{FP} \mid A_{\frac{\pm}{\mathcal{Q}}}^{\pm} f(\dots) \wedge f(\dots) \in \mathcal{A}(t_a)\}$$

⁶Otherwise we have either to accept that each assignment to some structured variables destroys all equalities involving other components of it or to implement some strategy (for example copying) preserving useful and safe access terms.

(“roots” of memory access terms in $\mathcal{A}(t_a)$). Notice that flow-insensitive approximations of alias information may cause conservative results of **ERA**. Therefore **ERA** and alias analyses used in its implementation should have the same sensitivity to the control flow.

Assignment statement $L[v := exp]$.

1. In the state L evaluate exp using evaluation transformer formally defined earlier in **Section 1.3.2**. Let L' be a result of the evaluation and let E be a nonterminal such that $E \Rightarrow exp$.
2. Perform $L' \Downarrow \overline{\mathcal{A}(v, L')}$. Do not remove E .
3. Add the term v so that $E \Rightarrow v$.

Unfortunately, in some cases this abstraction of the assignment statement fails as before. For example, this assignment transformer corresponding to $\mathbf{x} := \mathbf{x} + 1$ and being applied to the state $L = \{(x > 0) = TRUE\}$ gives only the trivial identity $L[x := x + 1] = \{x = x\}$. To improve accuracy of the analysis in these cases we can consider “artificial” variables associated with scalar variables of the program which will store previous values of the original ones. Under this approach between first and second steps of the assignment statement effect definition we should insert the step

... Let $A \rightarrow v$ and $B \rightarrow v'$ where v' is associated with v . Remove the second rule and add $A \rightarrow v'$ if it is needed.

Under this approach we shall have $L[x := x + 1] = \{(x' > 0) = TRUE, x = x' + 1\}$ from where we can deduce that $x > 0$ also.

2.2 Other transformers

- *Program.*

Given a program

```

PROGRAM;
  VAR  $x : T$ ; (* variables *)
BEGIN
   $S$  (* statements *)
END.

```

we can define the following transformer corresponding to it

$$\perp' \llbracket PROGRAM \rrbracket = \perp' \llbracket x := \omega \rrbracket \llbracket S \rrbracket$$

where ω represents the indefinite value. Notice that ω is not a constant.

- *Empty statement.*

$$L \llbracket \] = L$$

- *Sequence of statements.*

$$L \llbracket S_1; S_2 \rrbracket = L \llbracket S_1 \rrbracket \llbracket S_2 \rrbracket$$

- *Read statement.*

$$L[\llbracket READ(x) \rrbracket] = L\{\!\{x\}\!\} \downarrow \overline{A(x)}$$

Notice that if for *read statement* as well as for other statements some set of user's pre- or post-assertions represented in the form of equalities of program terms is supplied then the analyzer can take them into consideration to check consistency and to include in the current computation state.

- *Write statement.*

$$L[\llbracket WRITE(x) \rrbracket] = L\{\!\{x\}\!\}$$

- *Conditional statement.*

IF **p** **THEN** **S_t** **ELSE** **S_f** **END.**

If $L' = L\{\!\{p\}\!\}$ then

$$L[\llbracket IF \rrbracket] = L'\{\!\{p \equiv TRUE\}\!\} \llbracket S_t \rrbracket \cap L'\{\!\{p \equiv FALSE\}\!\} \llbracket S_f \rrbracket.$$

- *Cycle statement.*

CYCLE

S (* body of cycle *)

END

where **S** is a composed statement that possibly contains occurrences of exit-of-cycle statements **EXIT_k**. When the sequence

$$L_0 = L, \quad L_n = L_{n-1}[\llbracket S \rrbracket] \quad \text{for } n > 0$$

becomes stabilize

$$L[\llbracket CYCLE \rrbracket] = \cap_k E_k$$

where E_k is a stationary entry state for **EXIT_k**. If this process does not become stabilize then some widening operator should be used (see Section 3).

- *Halt, exit and return statements.*

$$L[\llbracket EXIT \rrbracket] = L[\llbracket RETURN \rrbracket] = \top'$$

- *Call of function.* We assume that return of results of function calls is implemented as an assignment to variables having the same names as invoked functions (connection with their call sites should be taken into account). The function bodies may contain **RETURN** statements as well.

FUNCTION **F(x: T₁) : T;**

VAR **y: T₂;** (* local variables *)

BEGIN

S (* statements *)

END.

$$L[\llbracket FUNCTION \rrbracket] = L[x := e; y := \omega; F := \omega] \llbracket S \rrbracket \cap R$$

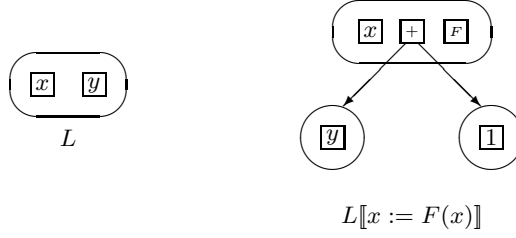


Figure 6: Function call for $F(a) = a + 1$.

where e is a factual parameter of the function (see the note for **READ** statement) and $R = \cap R_k$ is intersection of stationary entry states for return statements in **S** if they exist and $R = \top'$ otherwise. In the term being evaluated the result of a function call is represented by F .

3 Widening operator and convergence of the analysis

Our abstract semantic domain does not satisfy the (*descending*) *chain condition* and therefore it requires some *widening operator* [9, 10, 11]. To guarantee the convergence of the abstract interpretation we should use a dual widening operator:

- $\forall x, y \in \wp(\mathcal{EQS}) \Rightarrow x \supseteq x \tilde{\nabla} y \text{ \& \& } y \supseteq x \tilde{\nabla} y$,
- for all decreasing chains $x_0 \supseteq x_1 \supseteq \dots$, the decreasing chains defined by $y_0 = x_0, \dots, y_{i+1} = y_i \tilde{\nabla} x_{i+1}, \dots$ are not strictly decreasing.

The iteration sequence with widening is convergent and its limit is a sound approximation of the fixpoint.

3.1 Widening operator for ERA-grammars

Infinite chains can appear because corresponding languages have common infinite subsets generated by cyclic derivations in grammars. The source of that in **ERA** is **term identification**. We can avoid this problem by imposing the constraint that grammars must be acyclic. Within the semilattice $\wp(\mathcal{EQS})(\supseteq, \mathcal{EQS}, \cap)$, the subsemilattice of finite languages⁷ generated by such grammars satisfies the chain condition, but such languages are not expressive enough. Our solution is the following. Grammars are not originally restricted but if in course of abstract interpretation the grammar size becomes greater than some parameter, then “harmful” cycles must be destroyed. To this end we remove grammar rules participating in cyclic derivations. Correctness of this approximation of intersection follows from **Lemma 2**.

⁷Notice that sets of term equalities of a special form corresponding to these languages were used by V. Sabelfeld to develop effective algorithms of recognizing equivalence for some classes of program schemata [39, 40].

Detecting such rules is no simpler than the “minimum-feedback-arc/vertex-set” problem (**MFAS** or **MFVS**) if we consider the grammars as directed graphs. These sets are the smallest sets of arcs or vertices, respectively, whose removal makes a graph acyclic. We suppose that the “feedback vertices” choice is more natural for our purposes. In the general case this problem is NP -hard, but there are approximate algorithms that solve this problem in polynomial [41] or even linear [37] time. Consideration of weighted digraphs makes it possible to distinguish grammar rules with respect to their worth for accuracy of the analysis algorithm. However, perspectives of this approach are not clear now for complexity/precision reasons of such algorithms. For example, [19] proposes an algorithm for weighted **FVS**-problem requiring $O(n^2\mu(n)\log^2 n)$ time where $\mu(n)$ is complexity of matrix multiplication.

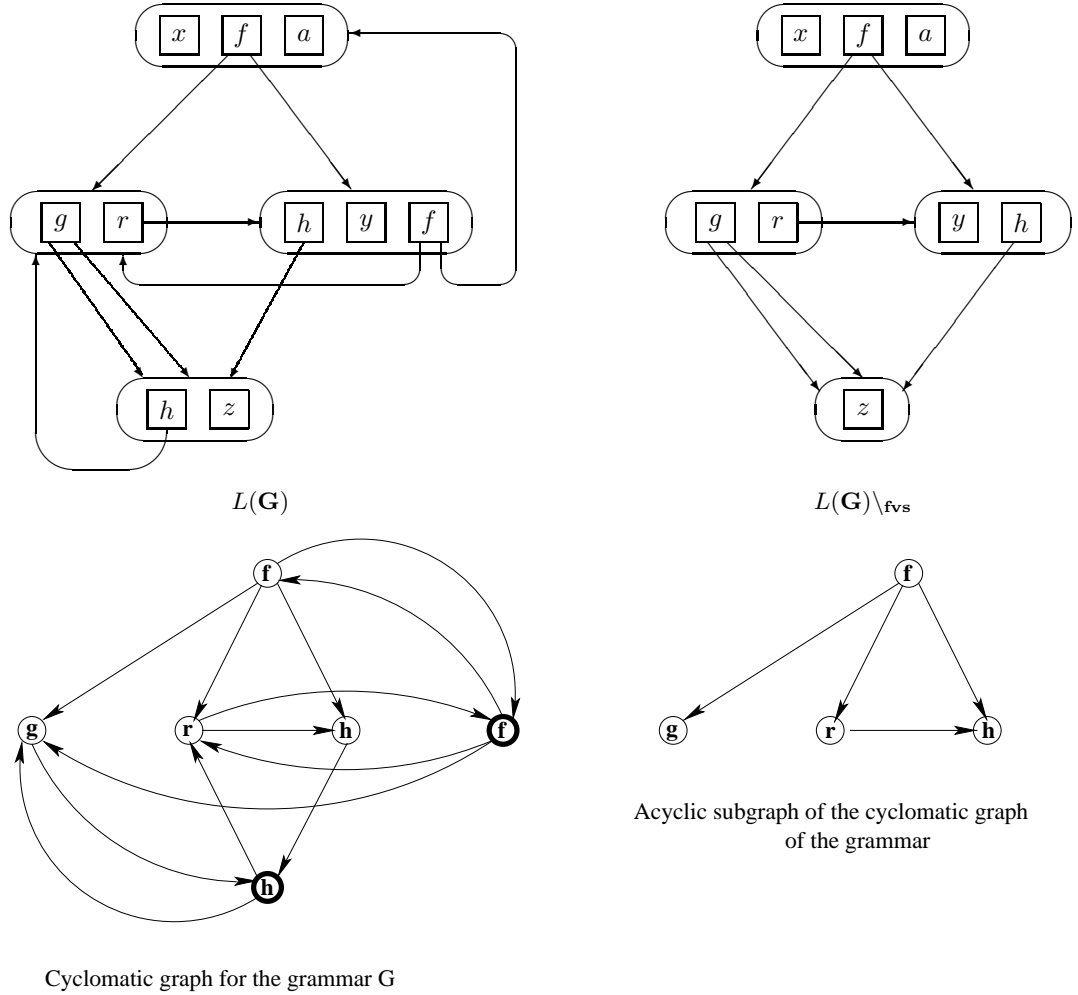


Figure 7: **FVS**-transformation.

Our widening operator for the analysis of equality relationships is defined in the following way. A vertex set of the cyclomatic graph⁸ of the grammar \mathbf{G} is a set of functional symbols existing in \mathbf{G} . An arc (f, g) belongs to its arc set if \mathbf{G} contains rules $A \rightarrow f(\dots, B, \dots)$ and $B \rightarrow g(\dots)$. A transformation of a cyclomatic graph which involves detecting some **FVS** (it can be an upper approximation of a minimal feedback set) and removing all vertices from the **FVS** is said to be an **FVS**-transformation (an example is shown in **Figure 7**). Let $L \setminus_{\mathbf{FVS}}$ be a language obtained from L by **FVS**-transformation applied to the grammar generating L . We define

$$L(\mathbf{G}_1) \widetilde{\nabla} L(\mathbf{G}_2) = \begin{cases} L(\mathbf{G}_1) \setminus_{\mathbf{FVS}} \cap L(\mathbf{G}_2) & \text{if } |\mathbf{G}_2| \geq |\mathbf{G}_1| > d, \\ L(\mathbf{G}_1) \cap L(\mathbf{G}_2) \setminus_{\mathbf{FVS}} & \text{if } |\mathbf{G}_1| > |\mathbf{G}_2| > d, \\ L(\mathbf{G}_1) \cap L(\mathbf{G}_2) & \text{otherwise,} \end{cases}$$

where d is a user-defined parameter. It is reasonable to choose this parameter, depending on number of variables of analyzed programs, as a linear function with a small factor of proportionality. Notice that in this case the lengths of appearing chains linearly depend on number of variables living simultaneously.

3.2 Divergence of the analysis

Is the widening operator, being rather complex, really needed for the analysis of equality relationships? Are there programs which, being analyzed, generate infinite chains of semantic properties? It should be mentioned that constructing such program examples has been a problem for a long time. In [18] we stated our belief that their existence seems hardly probable. These attempts failed, because they concentrated on constructing an example with completely non-interpretable functional symbols, i.e. in the frame of the “pure” theory of program schemata.

As already noticed, we can widely vary the interpretability of the analysis algorithm. In order to construct an example, it will suffice to use the following rule of completion:

$$\text{if } (t_1 = t_2) = TRUE \in L \text{ then } \mathbf{Co}(L) = L\{t_1 \equiv t_2\}.$$

We consider the following example:

<pre> ... x:=f(y); if f(x)=f(y) then while y=f(g(y)) do y:=g(y) end end ... </pre>	<pre> ... x:=sign(y); if sign(x)=sign(y) then while y=sign(abs(y)) do y:=abs(y) end end ... </pre>
<p>program scheme</p>	<p>“real-world” program</p>

The properties computed at the body entry belong to an infinite decreasing chain. On **Figure 8** a state L_e describes properties valid before cycle execution; states L_1

⁸A graph represents cyclic derivations in the grammar. The author did not find another appropriate name for this object in the Computer Science and Discrete Mathematics literature.

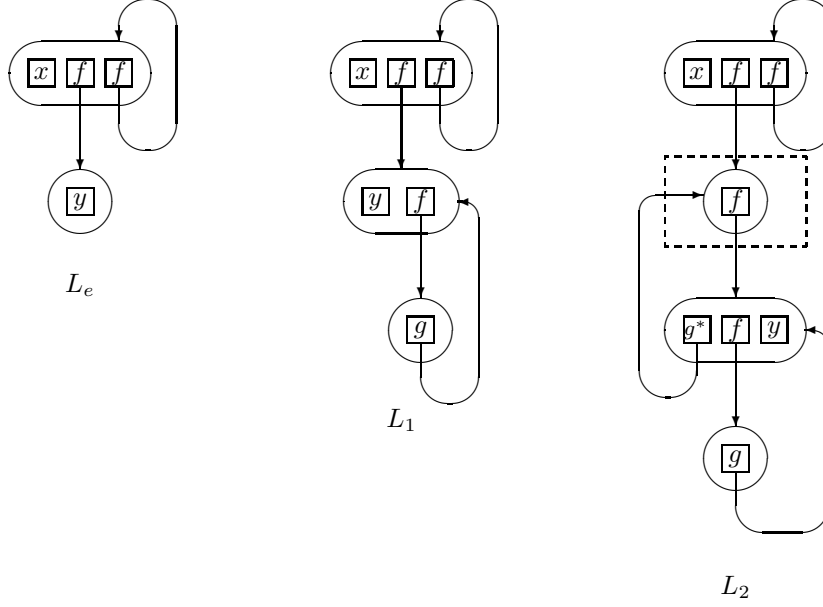


Figure 8: The case of divergence of the analysis.

and L_2 describe properties at the entry of the cycle body for first and second iteration, respectively. It is easy to see that $L_1 \cap L_2$ coincides with L_2 except in the equality relationships containing terms generated by g^* . Therefore every time we obtain the next state, the functional element g^* is absent and the sub-net placed in the dashed box will repeat more and more times.

To obtain a “real-world” program from this program scheme, we interpret the functional symbols in the following manner: $f \equiv \text{sign}$ and $g \equiv \text{abs}$. We would like to point out the following interesting property. Execution of this piece of code (i.e., its behavior determined with the standard semantics) diverges only for two values of y : 0 and 1. At the same time the analysis algorithm (i.e. execution of the piece of code under our nonstandard semantics) is always divergent on condition that a widening operator is not used and the assumption on interpretation mentioned above holds.

Is this program actually real-world? The reader should decide that by himself but we would like to underline the following⁹. On the one hand, the interpretability of the analysis algorithm can be varied in wide ranges and, on the other, we are not able to prove formally the impossibility of such behavior of the analyzer under the considered interpretation. So, we can choose either a lean analysis using acyclic grammars only or another one using arbitrary grammars and some widening operator.

⁹The penetrating reader may notice that this program to be considered as human written is really stupid (it can be slightly intellectualized if we interpret g as “add 1”). But for automatic generators of programs such a code does not seem improbable.

3.3 Complexity of the analysis

In [18] we pointed out the following upper bound on time for the algorithm of **ERA**: $O(nmG_{max}^2)$, where n is program size, m is maximum of number of program variables existing at a time, and G_{max} is maximum of sizes of grammars appearing in course of the analysis. Due to our construction of widening operator (namely, choice of the parameter d linearly depended on m) we can assume that $G_{max} = O(m)$.

This bound can be deduced with help of **Theorem 6** in [5]. The theorem states that for the recursive strategy of chaotic iteration maximum complexity is

$$h \cdot \sum_{c \in C} \delta(c) \quad (\leq h \cdot |C| \cdot |W|), \quad (**)$$

where h is maximum length of increasing chains built by a widening operator, C is set of control program points, $\delta(c)$ is depth of the control point c in hierarchy of nested strongly connected components of the control flow graph containing c , and W is set of vertices where a widening operator is applied during the analysis.

For well-structured programs we can assume that maximum depth of nested loops does not depend on program size and is bounded by some constant. By (**) we conclude that number of algorithm steps does not exceed $O(nm)$. Since time complexity of all operations used in the analysis are estimated by G_{max}^2 we obtain $O(nmG_{max}^2)$ (or $O(nm^3)$) upper bound. Notice that to improve the results of the analysis it is possible to use rich semantic completion and more precise **FVS**-algorithms that have more than quadratic time complexity.

However, experimental results show that an approximation of a fixed point for the heads of cycle bodies is usually attained after at most two iterations and time complexity of the analysis is proportional to nG_{max} . Also, the user can turn off checking a threshold after which widening is started. In this case, he (consciously) admits some chance that the analysis diverges but we believe that this chance is not too big.

It is easy to see that the space complexity of the equality relationship analysis is $O(nG_{max})$ and it is essentially depended on the number of variables. We estimate the actual space requirements as 1.5–2.0 Mb per 1000 program lines for middle-size programs.

4 Processing of invariants and experimental results

4.1 Usage of ERA-invariants

ERA produces some set of invariants involving program terms which can be useful at different steps of program development and processing: debugging, verification (for that the invariants are interesting themselves), specialization, and optimization. The automatic prover mentioned above can be used at step of post-processing results of the analysis.

We notice that the analyzer can tell the user useful information both at the stage of analyzing (this means that it is possible that there exist execution traces where such computational states appear; we mark with + properties which can be detected at this

stage) and at the stage of processing of results of the analysis (these properties hold for each execution trace leading to this program point). We shall briefly list some program properties that can be extracted from computation states L_{in} and L_{out} being for a statement **S** the input and output states respectively.

- + *Variable x has an indefinite value* if L_{in} contains $x = \omega$.
- + *Error in evaluation of an expression*: division by zero, out of type ranges, nil-pointer dereferencing, etc.
- ***S** is inaccessible* if $L_{in} = \top'$. This information can correspond to different properties of program execution: potentially infinite cycles and recursive calls, dead branches of conditional statements, useless procedural definitions, etc.
- *Assignment statement $\mathbf{v} := \mathbf{exp}$ is redundant* if L_{out} contains $v = v'$ where v' is the variable associated with v .
- *Unused definitions* (constants, variables, types).
- *Constant propagation*. Notice that **ERA** can detect that an expression is constant not only when constants for all variables in this expression are known.
- More general: for some expression *there exists an expression that is equal to the original one and calculated more efficiently* (with respect to an given criterion time/space and target computer architecture).

Obviously, this list is not complete and there are many other properties which can be extracted from the invariants. For example, we can consider systems of equations/inequalities contained in the gathered invariants and try to solve them to derive more precise ranges for values of expressions or the inconsistency of this computational state.

Apart from the automatic mode when invariants are processed automatically, we provide an interactive mode to visualize results of our analysis in a hypertext system. *HyperCode* presented in [7, 3] is an open tunable system for visualization of properties of syntactic structures. There are two cases: visualization of all properties detected in the automatic mode and the user-driven processing and visualization of properties.

The experiments show that not all program properties of interest can be automatically extracted out of the computed invariants. It is not judicious to consider many particular cases and to hardly embed them into the system. Instead, the system facilitates the specification of the user request with some friendly interface. He chooses a program point and an expression and obtains those and only those equality relationships, valid at this point, where this expression occurs as a super- or sub-term.

4.2 Program examples

An example of a program is presented below. The properties detected by the analyzer are indicated in comments.

program	length (lines)			size (bytes)		
	<i>M2Mix</i>	ERA	improv.	<i>M2Mix</i>	ERA	improv.
KMP	167	133	20.35%	2996	2205	26.4%
Lambert	361	326	9.7%	6036	2564	57.5%
Automaton	37	35	5.4%	969	926	4.5%
Int_{Fib}	87	77	11.5%	1647	1432	13.05%
Ackerman	64	62	3.1%	1384	1322	4.5%
	average		10.01%	average		21.19%

Table 2: Comparison **ERA** and *M2Mix*.

```

var x,y,z: integer;
procedure P(a,b: integer):integer;
begin                                (*parameters are always equal*)
    return a+b                        (*expression can be simplified: 2*a*)
end P;
begin
    Read(x);
    while x≤0 do
        Read(x);
        x := x+1;
        z := x+z;                    (*variable z might be uninitialized*)
        y := x+1;
        if x=0 then
            z := y;                  (*r-value can be simplified: z:=1*)
        else
            z := x+1;                (*r-value can be simplified: z:=y*)
            x := y;
        end;
        Write( P(y,z) )             (*call can be transformed: Write(2*y)*)
    end;
    x := z div (y - z);              (*arithmetical error*)
    Write(x)                         (*inaccessible point*)
end.

```

On basis of the analysis, this program can be transformed into the following:

```

var x:integer;
begin
    Read(x);
    while x≤0 do
        Read(x); x := x+2; Write(2*x)
    end
    ERROR_EXCEPTION;
end.

```

In **Table 2** we present some results of optimization based on our analysis of residual programs generated by *M2Mix* specializer [6, 33]. To compile these examples, we

used XDS Modula/Oberon compiler v.2.30 [20]. The following programs have been investigated:

- **KMP** — the “naïve” matching algorithm specialized with respect to some pattern; the residual program is comparable to Knuth, Morris, and Pratt’s algorithm in efficiency. (see also **Appendix**).
- **Lambert** — a program drawing Lambert’s figure and specialized with respect to number of points.
- **Automaton** — an interpreter of a deterministic finite-state automaton specialized with respect to some language.
- **Int_{Fib}** — an interpreter of MixLan [33] specialized with respect to a program computing Fibonacci numbers.
- **Ackerman** — a program computing some values of Ackerman’s function and specialized with respect to its first argument.

Let us comment briefly on the obtained results. Reducing length of a program can be considered as reducing number of operators and declarations. In these examples the optimizing effect was typically attained by the removal of redundant assignments, dead operators, unused variables and the reduction of operator strength. The only exception is **KMP** program characterized by high degree of polyvariance (roughly speaking it means presence of deep-nested conditional statements) and an active usage of array references. Here some IF-statements with constant conditions and redundant range checks were eliminated. Notice that the last optimizing transformation is very important for Modula-like languages where such checks are defined by the language standard. Such notable optimizing effect for the **Lambert** program is explained by deep reduction of power of floating-point operations which cannot be achieved by optimizing techniques now used in compilers. Since **Automaton** and **Ackerman** programs are quite small, their optimization gives conservative results. However, they would be better for the **Ackerman** program if the implementation of **ERA** were context-sensitive. Substantial speed-up of these optimized programs was not obtained (it was less than 2%) and this is not surprising since the great bulk of specializers take it as a criterion of optimality.

These experiments show that an average reduction of size of residual programs is 20–25%. Because the case of **KMP** program seems to be the most realistic¹⁰, we suppose that such improvement can be achieved in practice for real-world programs and it will be increased for large residual programs with a high degree of polyvariance and active usage of arrays and float-point arithmetics. It is the author opinion that the analysis of automatically generated (from high-level specifications as well) programs is the most promising direction of its application, especially in the context-sensitive implementation of **ERA**.

¹⁰Unfortunately our experiments were not exhaustive enough since the partial evaluation is not involved yet in real technological process of the software development and hence finding large residual programs is a hard problem.

Acknowledgement

The author wishes to thank M.A. Bulyonkov, P. Cousot, R. Cousot, and V.K. Sabelfeld for support, useful discussions and remarks.

References

- [1] A. Aho and J. Ullman. *The theory of parsing, translation and compilation*, volume 1. Prentice-Hall Inc., 1972.
- [2] B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *Proc. of the 15th Annual ACM Symposium Principles of Programming Languages*, pages 1–11. ACM Press, 1988.
- [3] D. Baburin, M. Bulyonkov, P. Emelianov, and N. Filatkina. Visualization facilities in program reengineering. *Programmirovaniye*, 27(2):21–33, 2001.
- [4] L. Berman and G. Markowsky. Linear and non-linear approximate invariants. Research Report RC7241, T.J. Watson Research Center of IBM, Yorktown, NY, 1976.
- [5] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the International Conference on Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 129–141. Springer-Verlag, 1993.
- [6] M. Bulyonkov and D. Kochetov. Practical aspects of specialization of Algol-like programs. In *Proc. of the International Seminar on Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 1996.
- [7] M. Bulyonkov and D. Kochetov. Visualization of program properties. Research Report 51, Institute of Informatics Systems, Novosibirsk, Russia, 1998.
- [8] R. Clarisó and J. Cortadella. The octahedron abstract domain. In *Proc. of the 11th International Static Analysis Symposium*, volume 3148 of *Lecture Notes in Computer Science*, pages 312–327. Springer-Verlag, 2004.
- [9] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Rec. of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
- [10] P. Cousot and R. Cousot. Abstract interpretation and application to logic program. *Journal of Logic Programming*, 13(2–3):103–180, 1992.
- [11] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

- [12] P. Cousot and R. Cousot. Formal languages, grammar and set-constraint-based program analysis by abstract interpretation. In *Rec. of the Conference on Functional Programming Languages and Computer Architecture*, pages 170–181. ACM Press, 1995.
- [13] P. Cousot and R. Cousot. Software analysis and model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 37–56. Springer-Verlag, 2002.
- [14] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Records of the 5th annual ACM Symposium on Principles of Programming Languages*, pages 84–96. ACM, ACM Press, January 1978.
- [15] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k -limiting. *SIGPLAN Notices*, 29(6):230–241, 1994. Proc. of the ACM SIGPLAN’94 Conference on Program Language Design and Implementation.
- [16] P.G. Emelianov and D.E. Baburin. Semantic analyzer of Modula-programs. In *Proc. of the 4th International Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 361–363. Springer-Verlag, 1997.
- [17] P.G. Emelianov and V.K. Sabelfeld. Analyzer of semantic properties of Modula-programms. In *Software intellectualization and quality*, pages 100–107. Institute of Informatics Systems, Novosibirsk, Russia, 1994.
- [18] P.G. Emelianov. Analysis of the equality relation for the program terms. In *Proc. of the Third International Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 174–188. Springer-Verlag, 1996.
- [19] G. Even, J. Naor, B. Schieber, and M. Sudan. Approximating minimum feedback sets and multi-cuts. In *Proc. of the 4th International Conference on Integer Programming and Combinatorial Optimization*, volume 920 of *Lecture Notes in Computer Science*, pages 14–28. Springer-Verlag, 1995.
- [20] Excelsior LLC. Native xds-x86 modula-2/oberon-2 development toolset.
- [21] K. Gargi. A sparse algorithm for predicated global value numbering. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 45–56, 2002.
- [22] A.D. Gordon and A.M. Pitts, editors. *High Order Operational Techniques in Semantics*. Publications of Newton Institute. Cambridge University Press, 1998.
- [23] Ph. Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30:165–199, 1989.
- [24] Ph. Granger. Static analysis of linear congruence equalities among variables of a program. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 493 of *Lecture Notes in Computer Science*, pages 169–192. Springer-Verlag, 1991.

- [25] S. Gulwani and G.C. Necula. Discovering affine equalities using random interpretation. In *Proc. of the ACM SIGPLAN-SIGACT 2003 Principles of Programming Languages*, pages 74–84, 2003.
- [26] S. Gulwani and G.C. Necula. A polynomial-time algorithm for global value numbering. In *Proc. of the 11th International Static Analysis Symposium*, volume 3148 of *Lecture Notes in Computer Science*, pages 212–227. Springer-Verlag, 2004.
- [27] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.
- [28] Nevin Heintze, Joxan Jaffar, and Răzvan Voicu. A framework for combining analysis and verification. In *Records of the 27th Annual ACM Symposium on Principles of Programming Languages*, pages 26–39, 2000.
- [29] N. Heintze and J. Jaffar. Set constraint and set-based program analysis. In A. Borning, editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 281–298. Springer-Verlag, 1994.
- [30] N.D. Jones and S.S. Muchnick, editors. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
- [31] N.D. Jones. Flow analysis of lazy higher-order functional programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 103–122. Ellis Horwood, 1987.
- [32] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.
- [33] D.V. Kochetov. *Effective specialization of Algol-like programs*. Ph.D. Thesis, Institute of Informatics Systems, Novosibirsk, Russia, 1995.
- [34] F. Masdupuis. Array operations abstraction using semantic analysis of trapezoid congruences. In *Proceedings of the International Conference on Supercomputing*, pages 226–235. ACM Press, 1992.
- [35] F. Masdupuis. Semantic analysis of interval congruences. In *Proceedings of the International Conference Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 142–155. Springer-Verlag, 1993.
- [36] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Proceedings of the Second Symposium on Programs as Data Objects*, volume 2053, pages 155–169, 2001.
- [37] B.K. Rosen. Robust linear algorithms for cutsets. *Journal Algorithms*, 3:205–217, 1982.

- [38] O. Rüthing, J. Knoop, and B. Steffen. Detecting equalities of variables: Combining efficiency with precision. In *Proc. of the 6th International Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pages 232–247. Springer–Verlag, 1999.
- [39] V. Sabelfeld. Polynomial upper bound for the complexity of the logic-termal equivalence decision. *Doklady Akademii Nauk, Matematika*, 249(4):793–796, 1979.
- [40] V. Sabelfeld. The logic-termal equivalence is polynomial-time decidable. *Information Processing Letters*, 10(2):102–112, 1980.
- [41] E. Speckenmeyer. On feedback problems in digraphs. In *Proceedings of the 15th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 411 of *Lecture Notes in Computer Science*, pages 218–231. Springer–Verlag, 1990.
- [42] A. Venet. Automatic analysis of pointer aliasing for untyped programs. *Science of Computer Programming*, 35(2-3):223–248, 1999.
- [43] M. Wegman and F.K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Language and Systems*, 13(2):181–210, 1991.

Appendix. Analysis of KMP

The appendix presents results of application of **ERA** to **KMP** program generated by the specializer *M2Mix*. This program is a specialization of a program implementing the naïve pattern matching **Match(p,str)** with respect to the pattern **p="ababb"**. The invariants are written as comments at program points where they hold. We can conclude that:

- The target string necessarily ends with "#" and the variable **ls** is equal to the string length (line 10).
- Every time when some element of **str** (lines 20, 31, 42, 53, 58, 67, 72, 78, 86, 108, 113, 129, 140, 151, 162) is used in second **LOOP**, the value of its index expression does not exceed the value of the variable **ls**. The same is true for the value of a variable before the increment statements **INC(s)** (lines 23, 34, 45, 62, 81, 89, 94, 98, 103, 116, 120, 132, 143, 154, 165). Therefore, it suffices to check that a value of **ls** is not beyond the ranges determined by the type **_TYPE354a04** during input of the target string (line 8). So, in the second cycle all range checks can be eliminated.
- The assignment **_cfg_counter:=0** is redundant (line 24).
- Conditions **str[s+2]='a'** and **str[s]='a'** are always false (lines 78 and 86, respectively) because two different constants are equal. So, the code of **THEN**-branches is dead.
- The conditions at the lines 63, 68, 74, 82, 104, 109 are false, too. However, automatic detection of these properties are not as easily as the previous.

Using this semantic information, it is possible to build a new program functionally equivalent to **Match("ababb",str)**. In text of the program given below the underlined code can be eliminated.

```

MODULE Match;
  FROM FIO IMPORT File,Open,ReadChar,WriteInt,stdout;
  VAR _cfg_counter : CARDINAL; str_file : File;
  TYPE _TYPE354a04 = [0..20];
  TYPE _TYPE355004 = ARRAY _TYPE354a04 OF CHAR;
  VAR str : _TYPE355004; ls,s : _TYPE354a04;
BEGIN
  1: str_file := Open("target.dat");
  2: ls := 0;
  3: LOOP
  4:   str[ls] := ReadChar(str_file);
  5:   IF (str[ls]='#') THEN                                (*str[ls]='#')
  6:     EXIT
  7:   ELSE                                                  (*str[ls]≠'#')
  8:     INC(ls)
  9:   END
  10: END;                                                  (*str[ls]='#')
  11: s := 0;
  12: _cfg_counter := 0;                                    (*s = _cfg_counter = 0*)

```

```

13: LOOP
14:   CASE _cfg_counter OF
15:   | 0 : (*_cfg_counter = 0*)
16:       IF ((s+0) ≥ ls) THEN (*s ≥ ls*)
17:         WriteInt(stdout,(-1),0);
18:         EXIT
19:       END; (*0 < ls, s < ls*)
20:       IF (str[(s+0)]='a') THEN (*str[s] = 'a', 0 < ls, s < ls*)
21:         _cfg_counter := 1
22:       ELSE (*0 < ls, s < ls, str[0] ≠ 'a'*)
23:         INC(s); (*_cfg_counter = 0*)
24:         _cfg_counter := 0
25:       END (*0 < ls, s < ls*)
26:   | 1 : (*_cfg_counter = 1*)
27:       IF ((s+1) ≥ ls) THEN (*s + 1 ≥ ls*)
28:         WriteInt(stdout,(-1),0);
29:         EXIT
30:       END; (*s + 1 < ls*)
31:       IF (str[(s+1)]='b') THEN (*str[s + 1] = 'b'*)
32:         _cfg_counter := 2
33:       ELSE (*str[s + 1] ≠ 'b'*)
34:         INC(s); (*str[s] ≠ 'b', s < ls*)
35:         _cfg_counter := 0
36:       END
37:   | 2 : (*_cfg_counter = 2*)
38:       IF ((s+2) ≥ ls) THEN (*s + 2 ≥ ls*)
39:         WriteInt(stdout,(-1),0);
40:         EXIT
41:       END; (*s + 2 < ls*)
42:       IF (str[(s+2)]='a') THEN (*str[s + 2] = 'a'*)
43:         _cfg_counter := 3
44:       ELSE (*str[s + 2] ≠ 'a'*)
45:         INC(s); (*str[s + 1] ≠ 'a', s + 1 < ls*)
46:         _cfg_counter := 4
47:       END
48:   | 3 : (*_cfg_counter = 3*)
49:       IF ((s+3) ≥ ls) THEN (*s + 3 ≥ ls*)
50:         WriteInt(stdout,(-1),0);
51:         EXIT
52:       END; (*s + 3 < ls*)
53:       IF (str[(s+3)]='b') THEN (*str[s + 3] = 'b'*)
54:         IF ((s+4) ≥ ls) THEN (*s + 4 ≥ ls, s + 3 < ls*)
55:           WriteInt(stdout,(-1),0);
56:           EXIT
57:         END; (*s + 4 < ls*)
58:         IF (str[(s+4)]='b') THEN (*str[s + 3] = str[s + 4] = 'b'*)
59:           WriteInt(stdout,s,0);
60:           EXIT
61:         ELSE (*str[s + 3] = 'b', str[s + 4] ≠ 'b', s + 4 < ls*)
62:           INC(s); (*str[s + 2] = 'b', str[s + 3] ≠ 'b', s + 3 < ls*)
63:           IF ((s+0) ≥ ls) THEN (*s ≥ ls, s + 3 < ls*)
64:             WriteInt(stdout,(-1),0);
65:             EXIT
66:           END;
67:           IF (str[(s+0)]='a') THEN
68:             IF ((s+1) ≥ ls) THEN
69:               WriteInt(stdout,(-1),0);
70:               EXIT
71:             END;

```

72:	IF (str[(s+1)]='b') THEN	$(*str[s] = 'a', str[s+1] = 'b', str[s+2] = 'b',$
73:		$str[s+3] \neq 'b', s+3 < ls*)$
74:	IF ((s+2) ≥ ls) THEN	$(*s+2 \geq ls, s+3 < ls*)$
75:	WriteInt(stdout,(-1),0);	
76:	EXIT	
77:	END;	$(*str[s] = 'a', str[s+1] = 'b', str[s+2] = 'b',$
		$str[s+3] \neq 'b', s+3 < ls*)$
78:	IF (str[(s+2)]='a') THEN	$(*inaccessible point*)$
79:	_cfg_counter := 3	
80:	ELSE	$(*str[s] = 'a', str[s+1] = 'b', str[s+2] = 'b',$
		$str[s+3] \neq 'b', s+3 < ls*)$
81:	INC(s);	$(*str[s-1] = 'a', str[s] = 'b', str[s+1] = 'b',$
		$str[s+2] \neq 'b', s+2 < ls*)$
82:	IF ((s+0) ≥ ls) THEN	$(*s \geq ls, s+2 < ls*)$
83:	WriteInt(stdout,(-1),0);	
84:	EXIT	
85:	END;	$(*str[s-1] = 'a', str[s] = 'b', str[s+1] = 'b',$
		$str[s+2] \neq 'b', s+2 < ls*)$
86:	IF (str[(s+0)]='a') THEN	$(*inaccessible point*)$
87:	_cfg_counter:=14	
88:	ELSE	$(*str[s-1] = 'a', str[s] = 'b', str[s+1] = 'b',$
		$str[s+2] \neq 'b', s+2 < ls*)$
89:	INC(s);	$(*str[s-2] = 'a', str[s-1] = 'b', str[s] = 'b',$
		$str[s+1] \neq 'b', s+1 < ls*)$
90:	_cfg_counter:=4	
91:	END	
92:	END	$(*str[s-2] = 'a', str[s-1] = 'b', str[s] = 'b',$
		$str[s+1] \neq 'b', s+1 < ls*)$
93:	ELSE	$(*str[s] = 'a', str[s+1] \neq 'b', str[s+2] = 'b',$
		$str[s+3] \neq 'b', s+3 < ls*)$
94:	INC(s);	$(*str[s-1] = 'a', str[s] \neq 'b', str[s+1] = 'b',$
		$str[s+2] \neq 'b', s+2 < ls*)$
95:	_cfg_counter := 12	
96:	END	
97:	ELSE	$(*str[s] = 'a', str[s+2] = 'b', str[s+3] \neq 'b',$
		$s+3 < ls*)$
98:	INC(s);	$(*str[s-1] = 'a', str[s+1] = 'b', str[s+2] \neq 'b',$
		$s+2 < ls*)$
99:	_cfg_counter := 12	
100:	END	
101:	END	
102:	ELSE	$(*str[s+3] \neq 'b', s+3 < ls*)$
103:	INC(s);	$(*str[s+2] \neq 'b', s+2 < ls*)$
104:	IF ((s+0) ≥ ls) THEN	$(*s \geq ls, s+2 < ls*)$
105:	WriteInt(stdout,(-1),0);	
106:	EXIT	
107:	END;	$(*str[s+2] \neq 'b', s+2 < ls*)$
108:	IF (str[(s+0)]='a') THEN	$(*str[s] = 'a', str[s+2] \neq 'b', s+2 < ls*)$
109:	IF ((s+1) ≥ ls) THEN	$(*s+1 \geq ls, s+2 < ls*)$
110:	WriteInt(stdout,(-1),0);	
111:	EXIT	
112:	END;	$(*str[s] = 'a', str[s+2] \neq 'b', s+2 < ls*)$

```

113:          IF (str[(s+1)]='b') THEN      (*str[s]='a', str[s+1]='b', str[s+2]≠'b',
                                           s+2 < ls*)
114:              _cfg_counter := 2
115:          ELSE                             (*str[s]='a', str[s+2]≠'b', s+2 < ls*)
116:              INC(s);                     (*str[s-1]='a', str[s+1]≠'b', s+1 < ls*)
117:              _cfg_counter := 10
118:          END
119:      ELSE                             (*str[s+2]≠'b', s+2 < ls*)
120:          INC(s);                         (*str[s+1]≠'b', s+1 < ls*)
121:          _cfg_counter := 10
122:      END
123:  END
124:  | 4 :                                (*_cfg_counter = 4*)
125:      IF ((s+0)≥ls) THEN                 (*s ≥ ls*)
126:          WriteInt(stdout,(-1),0);
127:          EXIT
128:      END;                             (*s < ls*)
129:      IF (str[(s+0)]='a') THEN           (*str[s]='a', s < ls*)
130:          _cfg_counter := 1
131:      ELSE                             (*str[s]≠'a', s < ls*)
132:          INC(s);                       (*str[s-1]≠'a', s ≤ ls*)
133:          _cfg_counter := 0
134:      END
135:  | 10:                                (*_cfg_counter = 10*)
136:      IF ((s+0)≥ls) THEN                 (*s ≥ ls*)
137:          WriteInt(stdout,(-1),0);
138:          EXIT
139:      END;                             (*s < ls*)
140:      IF (str[(s+0)]='a') THEN           (*str[s]='a', s < ls*)
141:          _cfg_counter := 1
142:      ELSE                             (*str[s]≠'a', s < ls*)
143:          INC(s);                       (*str[s-1]≠'a', s ≤ ls*)
144:          _cfg_counter := 0
145:      END
146:  | 12:                                (*_cfg_counter = 12*)
147:      IF ((s+0)≥ls) THEN                 (*s ≥ ls*)
148:          WriteInt(stdout,(-1),0);
149:          EXIT
150:      END;                             (*s < ls*)
151:      IF (str[(s+0)]='a') THEN           (*str[s]='a', s < ls*)
152:          _cfg_counter := 14
153:      ELSE                             (*str[s]≠'a', s < ls*)
154:          INC(s);                       (*str[s-1]≠'a', s ≤ ls*)
155:          _cfg_counter := 4
156:      END
157:  | 14:                                (*_cfg_counter = 14*)
158:      IF ((s+1)≥ls) THEN                 (*s+1 ≥ ls*)
159:          WriteInt(stdout,(-1),0);
160:          EXIT
161:      END;                             (*s+1 < ls*)
162:      IF (str[(s+1)]='b') THEN           (*str[s+1]='b', s+1 < ls*)
163:          _cfg_counter := 2
164:      ELSE                             (*str[s+1]≠'b', s+1 < ls*)
165:          INC(s);                       (*str[s]≠'b', s < ls*)
166:          _cfg_counter := 4
167:      END
168:  END
END Match.

```